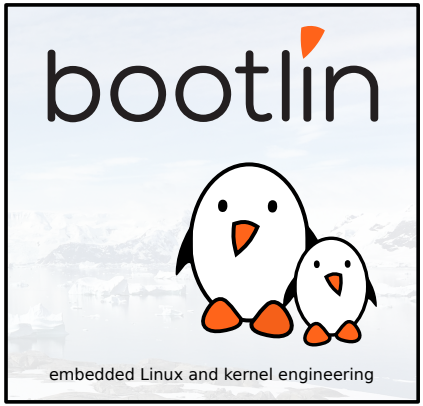
Timing Boot Time Reduction Techniques

Michael Opdenacker
michael.opdenacker@bootlin.com



Michael Opdenacker

- Founder and Embedded Linux engineer at Bootlin
  - Embedded Linux **expertise**
  - **Development**, consulting and training
  - Strong open-source focus
- Long time interest in embedded Linux boot time, and one of its prerequisites: small system size.
- Living in **Orange**, France.

Introduction

The system to optimize - Hardware

- Beagle Bone Black board (ARM Cortex A8 from TI)
- Beagle Bone LCD cape `https://elinux.org/Beagleboard:BeagleBone_LCD4`
- Standard USB webcam (supported through the `uvcvideo` driver.
- Booting on standard SDHC uSD card (cat 4) from Kingston

The system to optimize - Software

- Root filesystem built by Buildroot, just starting the `ffmpeg` program to show the video captured by the webcam.
- Initial boot time (boot to first decoded frame): 9.45s

Main goals for this presentation

- Not to cover all technical details of optimization techniques (will share useful resources at the end)
- But goal to benchmark the most useful techniques on a recent kernel.
- This should give you an idea about whether each technique is worth trying in your system or not.
- Last but not least, information harder to find about U-Boot's *Falcon Mode*. Apparently, there are still few people taking advantage of it, otherwise there would be more people talking about it.

Main optimization principles

- Start by optimizing things that won't reduce your ability for further measurement and optimizations
- This way, good to keep slower storage, and less efficient compression. This amplifies the phenomena to observe.
- Start from the last parts of boot time, and finish by the kernel and at the end, by the bootloader.

Premature optimization is the root of all evil.
Donald Knuth

Toolchain optimizations

ARM vs Thumb2

Compared the system compiled with ARM and with Thumb2

- Compiled with gcc 7.4, generating *ARM* code:
  Total filesystem size: 3.79 MB
  `ffmpeg` size: 227 KB
- Compiled with gcc 7.4, generating *Thumb2* code:
  Total filesystem size: 3.10 MB (-18 %)
  `ffmpeg` size: 183 KB (-19 %)
- Performance aspect: performance apparently slightly improved with Thumb2 (approximately less than 5 %, but there are slight variations in measured execution time, for one run to another).

musl vs uClibc

Tried to replace *uClibc* by *musl*

- musl library size: 680 KB (size of the `tar` archive for `lib/`)
- uClibc library size: 570 KB (-16 %)
- uClibc saves 110 KB (useful), but otherwise no other significant change in filesystem and code size. Not a surprise when the system is mostly filled with binaries relying on shared libraries.

We stuck to *uClibc*!

Application optimizations

General ideas

- Reduce the size of applications by recompiling them with only the features needed in your system. Run `./configure --help` for applications packaged by the autotools.
- Profile your application with `strace` or more advanced tools (`perf`) to optimize its behavior, especially if you authored the code.
- Reducing total size: inspect the root filesystem to find files that don't look necessary.

Results on ffmpeg

After rebuilding the system with the minimum `configure` options for `ffmpeg`:

- Total system size:
  From 16.11 MB to 3.54 MB (-78 %)
- Saves 150 ms in application loading + execution time
- Saves 120 ms in application execution time (file cache populated)
- Total boot time reduction: approximately 350 ms (faster mount time?)

Note: the size gains are massive. The performance gains are smaller, because Linux only loads the code actually used by the program. Some unnecessary code is eliminated though.

Init and root filesystem optimizations

Generic ideas

- Analyze system startup with `bootchartd` and eliminate unnecessary services
- Group required startup scripts into a single one (such as `/etc/init.d/rcS`), critical things first.
- Do not mount `/proc` and `/sys` if you don't need them (test your application), or mount them from C code.
- Simplify the BusyBox configuration to the minimum
- You could even start your application as the *init* process (`init` parameter in the kernel command line)
- Switching to static executables

Usefulness of root filesystem size reduction

- A smaller filesystem may be faster to mount
- Really useful when you boot your root filesystem as an *initramfs*:
  - Root filesystem archive embedded in the kernel binary
  - A smaller initramfs will make the kernel smaller to load from storage
  - Kernel decompression time will be faster too
  - The kernel and filesystem are loaded in a single read operation from storage, instead of multiple ones (more costly, more overhead)

Detect unnecessary files

Taking advantage of the fact that Linux filesystems record the last access time.

- Boot your system with the root filesystem mounted in read-write mode
- If your system has a correct time, find all files last accessed more than 5 minutes ago:
  `find / -atime +5 -type f`
- If your system doesn't have a correct time (and uses dates in 1970), back on your PC (if your storage is removable), you can find files last accessed less than 1000 minutes ago, when extracting the root filesystem:
  `find / -atime -1000 -type f`
- Remove such unwanted files with your build system (for example using `BR2_ROOTFS_POST_FAKEROOT_SCRIPT` in Buildroot)
- Also remove unnecessary directories with the same script.

Results: init and root filesystem optimizations

- Replaced all init scripts by a single script running the application
- Simplified BusyBox' configuration, only kept support for `echo`, `sh`, `sleep` and `test`, used in the single script. Size reduced from 682 KB to 86 KB.
- Eliminated unused files and directories
- Total filesystem size reduced from 3.54 MB to 2.33 MB (-34 %)
- Boot time difference: hardly noticeable, probably because init scripts didn't take much time.

Results: switching to static executables

- Making sense here as we only have two executables
- Removes all library code not used by such executables
- This allowed to remove more empty directories
- Total filesystem size reduced from 2.33 MB to 1.58 MB (-22 %)
- Boot time difference: about 20 ms, probably due to the time saved resolving and loading shared libraries.

```
.
[  44]  bin
   [129K]  busybox
   [   7]  echo -> busybox
   [   7]  hush -> busybox
   [   7]  sh -> busybox
   [   7]  sleep -> busybox
[   0]  dev
[ 305]  playvideo
[   6]  usr
   [  22]  bin
      [  17]  [ -> ../../bin/busybox
      [1.4M]  ffmpeg
      [  17]  test -> ../../bin/busybox

5 directories, 9 files
```

Filesystem optimizations

Switching to an initramfs

Multiple advantages:

- Possible after strong root filesystem size reduction
- Root filesystem that can embedded in the kernel image. Only one access to storage required (should be faster)
- No more need for block/storage and filesystem drivers. Smaller kernel, shorter load time and less initialization work.
- Only constraint: mount `devtmpfs` from userspace (no longer automatic)

Do not compress your initramfs

- If you ship your initramfs inside a compressed kernel image, don't compress it (enable `CONFIG_INITRAMFS_COMPRESSION_NONE`).
- Otherwise, by default, your initramfs data will be compressed twice, and the kernel will be bigger and will take a little more time to load and uncompress.
- Example on Linux 5.1 with a 1.60 MB initramfs (tar archive size) on Beagle Bone Black: this allowed to reduce the kernel size from 4.94 MB to 4.74 MB (-200 KB) and save about 170 ms of boot time.

Switching to initramfs - Results

- Initial impact on total boot time: about +300 ms, because of the bigger kernel and increased uncompression time.
- After disabling `CONFIG_BLOCK` and `CONFIG_MMC`, the compressed kernel size is reduced by 610 KB.
- Total boot time is even slighly lower than the initial value (-20 ms)

Kernel optimizations

Measure - Kernel initialization functions

To find out which kernel initialization functions are the longest to execute, add `initcall_debug` to the kernel command line. Here's what you get on the kernel log:

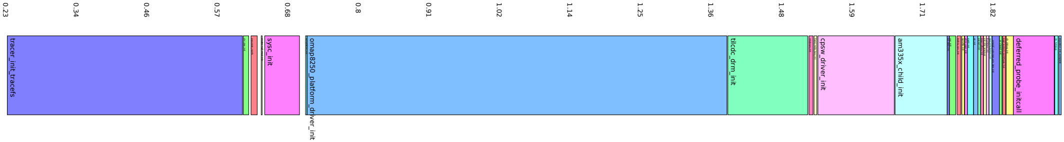
```
...
[    3.750000] calling  ov2640_i2c_driver_init+0x0/0x10 @ 1
[    3.760000] initcall ov2640_i2c_driver_init+0x0/0x10 returned 0 after 544 usecs
[    3.760000] calling  at91sam9x5_video_init+0x0/0x14 @ 1
[    3.760000] at91sam9x5-video f0030340.lcdheo1: video device registered @ 0xe0d3e340, irq = 24
[    3.770000] initcall at91sam9x5_video_init+0x0/0x14 returned 0 after 10388 usecs
[    3.770000] calling  gspca_init+0x0/0x18 @ 1
[    3.770000] gspca_main: v2.14.0 registered
[    3.770000] initcall gspca_init+0x0/0x18 returned 0 after 3966 usecs
...
```

You might need to increase the log buffer size with `CONFIG_LOG_BUF_SHIFT` in your kernel configuration. You will also need `CONFIG_PRINTK_TIME` and `CONFIG_KALLSYMS`.

Kernel boot graph

With initcall_debug, you can generate a boot graph making it easy to see which kernel initialization functions take most time to execute.

- Copy and paste the output of the dmesg command to a file (let's call it boot.log)
- On your workstation, run the scripts/bootgraph.pl script in the kernel sources:
  scripts/bootgraph.pl boot.log > boot.svg
- You can now open the boot graph with a vector graphics editor such as inkscape:



Using the kernel boot graph (1)

Start working on the functions consuming most time first. For each function:

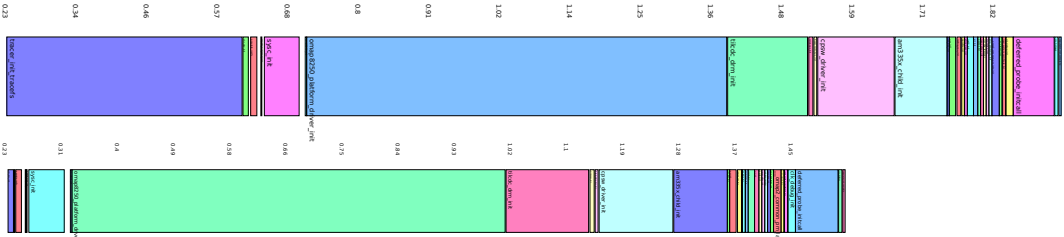
- Look for its definition in the kernel source code. You can use Elixir (for example `https://elixir.bootlin.com`).
- For unnecessary functionality, find which kernel configuration parameter compiles the code, by looking at the Makefile in the corresponding source directory.

Using the kernel boot graph (2)

- Postpone:
  - Find which module (if any) the function belongs to. Load this module later if possible.
  - Be careful: some function names don't exist, the names correspond to *modulename*`_init`. Then, look for initialization code in the corresponding module.
- Optimize necessary functionality:
  - Look for parameters which could be used to reduce probe time, looking for the `module_param` macro.
  - Look for delay loops and calls to functions containing `delay` in their name, which could take more time than needed. You could reduce such delays, and see whether the code still works or not.

Removing tracing infrastructure

- Corresponding to the `tracer_init_tracefs()` function call
- Unselected `Tracers` in `Kernel hacking` (enabled in our default configuration)
- Boot time savings: approximately 550 ms
- Kernel size savings: 217 KB



Other features to remove

- `omap8250_platform_driver_init (660 ms!)`
  Corresponds to the serial interface. No obvious reason found in the code. To be disabled later.
- `cpsw_driver_init` (112 ms)
  Corresponds to the network driver. Will be disabled.
- `am335x_child_init` (82 ms)
  Corresponds to the USB interface the camera is connected to. Cannot be skipped.
- All other items are too small. Some will be disabled by removing as many features as possible.

Preset loops per jiffy

- At each boot, the Linux kernel calibrates a delay loop (for the `udelay()` function). This measures a number of loops per jiffy (*lpj*) value. You just need to measure this once! Find the `lpj` value in the kernel boot messages:

```
Calibrating delay loop... 996.14 BogoMIPS (lpj=4980736)
```

- Now, you can add `lpj=<value>` to the kernel command line:

```
Calibrating delay loop (skipped) preset value.. 996.14 BogoMIPS (lpj=4980736)
```

- Tests on BeagleBone Bloack (ARM), Linux 5.1: -82 ms
  Time measured at the first kernel messages... the calibration loop is run before the message is issued.

Multiprocessing support (CONFIG_SMP)

- SMP is quite slow to initialize
- It is usually enabled in default configurations, even if you have a single core CPU (default configurations should support multiple systems).
- So make sure you disable it if you only have one CPU core.
- Results on BeagleBone Black:
  Compressed kernel size: -188 KB (-4.6 %)
  Total boot time: -126ms

Removing kernel module support

If possible!

- Before turning off `CONFIG_MODULES`, first manually turn off all features selected as modules. Otherwise, modules will be turned into build-ins while they were not necessary.
- Note that `menuconfig` and `xconfig` show you which modules can be removed or not because of dependencies.
- Before this, it is easier to remove all subsystems that you don't need for sure.
- Unselect features progressively and make copies for your configuration files. If the system no longer boots, you may have to start again from the beginning otherwise.
- Results: -82 KB in compressed kernel size, -20 ms of boot time.

Silencing the console, turning off kernel messages

- First, booting with the `quiet` command line parameter:
  Total boot time: -577 ms
- After removing `CONFIG_PRINTK` and `CONFIG_BUG`:
  Compressed kernel size: 1939152 (-118 KB, -5.8 %)
- After removing `CONFIG_KALLSYMS` too:
  Compressed kernel size: 1829168 (-107 KB, -5.7 %)
- Total savings:
  Compressed kernel size: -225 KB (-11 %)
  Total boot time: -767 ms

Using CONFIG_EMBEDDED and CONFIG_EXPERT

- Useful for keeping only the system calls that you're using in your system. This makes your kernel less generic, but that's fine when all applications are known in advance
- Compressed kernel size: -51 KB
- Total boot time: -34 ms

Compiling the kernel in Thumb2 mode

Can be selected by `CONFIG_THUMB2_KERNEL`

Results on BeagleBone Black:

- Compressed kernel size: +40 KB
- Total boot time: +5 ms

We keep compiling the kernel in ARM mode, unlike what we did for the root filesystem.

Choosing SLAB memory allocators

- SLAB: legacy, well proven allocator. Default choice on our board.
- SLOB: much simpler. More space efficient but doesn't scale well. Saves a few hundreds of KB in small systems (depends on `CONFIG_EXPERT`).
  Results on BeagleBone Black: -5 KB compressed kernel, +1,43 s total boot time!
- SLUB: more recent and simpler than SLAB, scaling much better (in particular for huge systems) and creating less fragmentation.
  Results on BeagleBone Black: +4 KB compressed kernel, + 2ms total boot time.

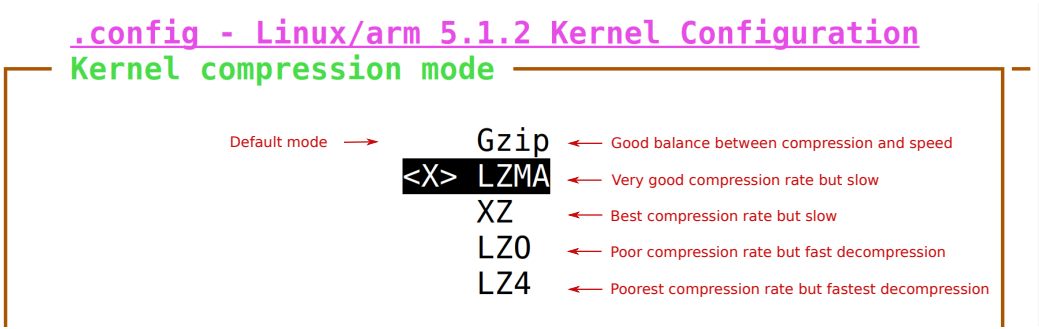
We're keeping SLAB!

```
Choose SLAB allocator

    SLAB
<X> SLUB (Unqueued Allocator)
    SLOB (Simple Allocator)
```

Kernel Compression

Depending on the balance between your storage reading speed and your CPU power to decompress the kernel, you will need to benchmark different compression algorithms. Also recommended to experiment with compression options at the end of the kernel optimization process, as the results may vary according to the kernel size.



Kernel compression options

Results on TI AM335x (ARM), 1 GHz, Linux 5.1

Timestamp	gzip	lzma	xz	lzo	lz4
Size	2350336	1777000	**1720120**	2533872	2716752
Copy	0.208 s	0.158 s	**0.154 s**	0.224 s	0.241 s
Time to userspace	1.451 s	2.167 s	1.999s	**1.416 s**	1.462 s

Gzip is close. It's time to try with faster storage (SanDisk Extreme Class A1)

Timestamp	gzip	lzma	xz	lzo	lz4
Size	2350336	1777000	**1720120**	2533872	2716752
Copy	0.150 s	0.114 s	**0.111 s**	0.161 s	0.173 s
Time to userspace	1.403 s	2.132 s	1.965 s	**1.363 s**	1.404 s

Lzo and Gzip seem the best solutions. Always benchmark as the results depend on storage and CPU performance.

Optimize kernel for size (1)

- `CONFIG_CC_OPTIMIZE_FOR_SIZE`: possibility to compile the kernel with `gcc -Os` instead of `gcc -O2`.
- Such optimizations give priority to code size at the expense of code speed.
- Results: the initial boot time is better (smaller size), but the slower kernel code can offset the benefits. Your system will run a bit slower!

Optimize kernel for size (2)

Results on BeagleBone Black, Linux 5.1, lzo compression

	O2	Os	Diff
Size	2533872	2390608	-5.7 %
Copy time	0.161 s	0.153 s	-8 ms
Starting kernel	0.912 s	0.904 s	-8 ms
Starting userspace	1.363 s	1.359 s	-4 ms
Total boot time	2.961 s	2.957s	-4 ms

Results on Microchip SAMA5D3 Xplained, Linux 3.10, gzip compression:

Timestamp	O2	Os	Diff
Starting kernel	4.307 s	4.213 s	-94 ms
Starting init	5.593 s	5.549 s	-44 ms
Login prompt	21.085 s	22.900 s	+ 1.815 s

Removing pseudo filesystems

- Remove the *proc* filesystem:
  Lzo compressed kernel size: -48 KB
  But ffmpeg doesn't work without *proc*
- Removing only proc options (`CONFIG_PROC_SYSCTL` and `CONFIG_PROC_PAGE_MONITOR`): -9 KB, no noticeable boot time change.
- Removing `CONFIG_CONFIGFS_FS`: -7 KB
- Removing the *sysfs* filesystem:
  Lzo compressed kernel size: -22 KB
  Time to userspace: -35 ms

Do it if compatible with your applications!

Other kernel optimizations

- Disable all *Compile-time checks and compiler options* in `Kernel hackingr`), in particular `CONFIG_DEBUG_INFO`:
  Compressed kernel size: -38 KB (-1.7 %)
  Time to userspace: -6 ms
- Replacing *ARM EABI stack unwinder* (`CONFIG_UNWINDER_ARM`) by the default mechanism:
  Compressed kernel size: -24 KB (-1.1 %)
  Time to user space: +0.8 ms, keeping it this option for its size savings.

Appending DTB to kernel

- Goal: group two copies from SD card:

```
0.862184 0.144682] 2205936 bytes read in 141 ms (14.9 MiB/s)
[0.873126 0.010942] 61284 bytes read in 7 ms (8.3 MiB/s)
```

- Using `CONFIG_ARM_APPENDED_DTB`, already enabled:

```
cat arch/arm/boot/zImage arch/arm/boot/dts/am335x-boneblack-lcd4.dtb > zImage
setenv bootcmd 'fatload mmc 0:1 81000000 zImage; bootz 81000000'
```

- Result: all bytes loaded at top speed

```
[0.863885 0.149569] 2266952 bytes read in 145 ms (14.9 MiB/s)
```

Starting kernel: 26 ms earlier
Userspace: 26 ms earlier

Bootloader optimizations

Bootloader optimizations: strategy

- It is definitely possible to improve performance in U-Boot. See our training slides: `https://bootlin.com/doc/training/boot-time/`
- However, the best solution is to **skip U-Boot**, using its *Falcon mode*:
  - We'll only execute the first stage of U-Boot, the SPL (Secondary Program Loader)
  - And will directly load the Linux kernel, instead of the U-Boot image. See `doc/README.falcon` in U-Boot sources for details.
- This is supported in the same way on all the boards with U-Boot support for SPL.

U-Boot Falcon mode - Preparation steps

Quite easy indeed!

- The Falcon mode needs a legacy kernel image:
  `make uImage LOADADDR=80008000`
  Copy it to the SD card.
- In U-Boot's `menuconfig` interface, go to the `SPL / TPL` menu and unselect `Support an environment`. Recompile U-Boot and update it on the SD card. In our tests, this saved 250 ms in Falcon mode!

See our training labs for details: `https://bootlin.com/doc/training/boot-time/`

U-Boot Falcon mode - without DTB (embedded in uImage)

- In the U-Boot command line, load the kernel image in RAM:
  `load mmc 0:1 81000000 uImage`
- Set the bootargs if needed:
  `setenv bootargs console=ttyO0,115200n8 rdinit=/playvideo lpj=4980736`
- Simulate booting the Linux kernel:
  `spl export atags 81000000`
- Using the address output by this command, store the atags information to MMC (`args` file):
  `fatwrite mmc 0:1 0x80000100 args 4000`
- Reset and the system should boot through only the U-Boot SPL

U-Boot Falcon mode - with DTB

- In the U-Boot command line, load the kernel image and DTB in RAM:
  ```
  load mmc 0:1 81000000 uImage
  load mmc 0:1 82000000 dtb
  ```
- Set the bootargs if needed:
  ```
  setenv bootargs console=ttyO0,115200n8 rdinit=/playvideo lpj=
  4980736
  ```
- Simulate booting the Linux kernel:
  ```
  spl export ftd 81000000 - 82000000
  ```
- Using the addresses output by this command, store the Flattened Device Tree (fdt) information to MMC (`args` file):
  ```
  fatwrite mmc 0:1 0x8ffd9000 args 1de57
  ```
- Reset and the system should boot through only the U-Boot SPL

U-Boot Falcon mode results

```
[0.000000 0.000000]
[0.000785 0.000785] U-Boot SPL 2019.01 (Oct 27 2019 - 08:04:06 +0100)
[0.057822 0.057822] Trying to boot from MMC1
[0.378878 0.321056] fdt_root: FDT_ERR_BADMAGIC
[0.775306 0.396428] Waiting for /dev/video0 to be ready...
[1.966367 1.191061] Starting ffmpeg
...
[2.412284 0.004277] First frame decoded
```

- Time to userspace: -479 ms
- Time to ffmpeg: -478 ms
- Total boot time: 2.412284

Challenges and issues

- Didn't manage to boot yet without the tty layer. At least the application doesn't start. According to the boot graph, expecting to save hundreds of ms.
- Waiting for 1.2 s for the USB camera to be enumerated.
  Any way around this USB asynchronousness issue?
- Solving these issues should allow to start displaying a video in less than 1 s.

```
[0.000000 0.000000]
[0.000785 0.000785] U-Boot SPL 2019.01 (Oct 27 2019 - 08:04:06 +0100)
[0.057822 0.057822] Trying to boot from MMC1
[0.378878 0.321056] fdt_root: FDT_ERR_BADMAGIC
[0.775306 0.396428] Waiting for /dev/video0 to be ready...
[1.966367 1.191061] Starting ffmpeg
...
[2.412284 0.004277] First frame decoded
```

Don't miss

- *Tuesday, October 29 - 11:30 - 12:05*
  We Need to Talk About Systemd:
  Boot Time Optimization for the New init daemon
  Chris Simmonds, 2net

Conference presentations and training materials

- Andrew Murray - The Right Approach to Minimal Boot Time (2010)
  Video: `https://frama.link/nrf696Hy` - Slides:
  `https://frama.link/uCBH9jQM`
  Great talk about the methodology.
- Chris Simmonds - A Pragmatic Guide to Boot-Time Optimization (2017)
  Video: `https://frama.link/Vnmj5t1m` - Slides:
  `https://frama.link/TC0YKM9N`
- Jan Altenberg - How to Boot Linux in One Second (2015)
  Video: `https://frama.link/BztbLy9T` - Slides:
  `https://frama.link/bFkvgLFR`
- Bootlin's Linux boot time training materials and labs:
  See our training slides: `https://bootlin.com/doc/training/boot-time/`

Most successful techniques in our project

Disable sysfs
- 35 ms
- 22 KB kernel size

Disable proc
- 48 KB kernel size

Concat kernel and DTB
- 22 ms

Kernel compression
-35 ms with LZO

Non standard kernel (EXPERT/EMBEDDED)
- 34 ms
- 51 KB kernel size

Disable tracing
- 550 ms
- 217 KB kernel size

Silent kernel
-767 ms
- 11% kernel size

uClibc instead of musl
-16 % library size

Disable modules
- 20 ms
- 82 KB kernel size

Delay loop calibration
-82 ms

Initramfs
- 20 ms

Apps with less options
- 350 ms
- 78 % system size

Disable SMP
- 126 ms
-4.6 % kernel size

Thumb2 toochain
-18 % code size

Static executables
- 20 ms
- 22 % system size

Uncompressed initramfs
- 170 ms
- 200 KB system size

Rootfs simplification
-34 % system size

U-Boot Falcon Mode
-478 ms

Questions? Suggestions? Comments?

Michael Opdenacker
michael.opdenacker@bootlin.com

https://bootlin.com/pub/conferences/2019/elce/